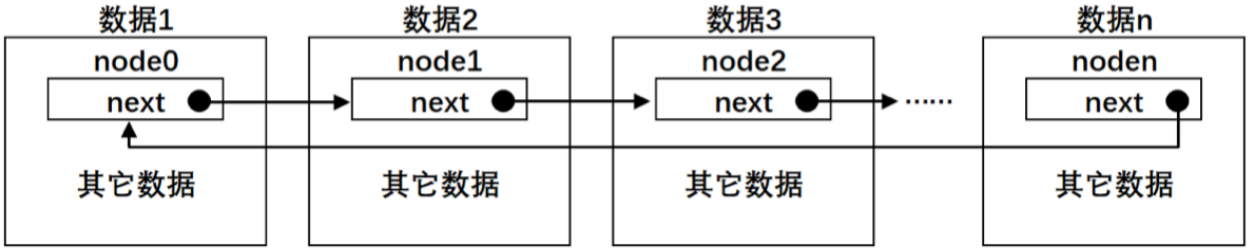
FreeRTOS(V10.02)内核源码梳理

I. FreeRTOS中的基本数据结构

A. 基础结构

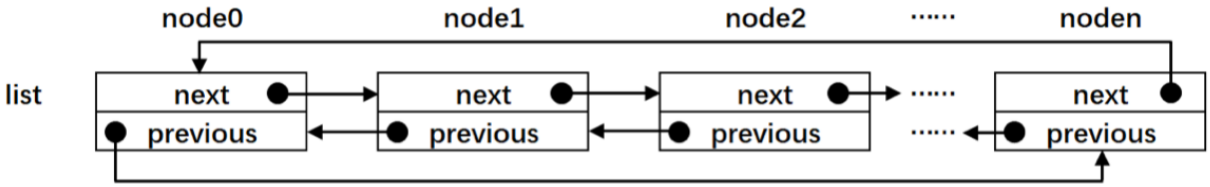
1. 单向链表

   该链表中共有n个节点，前一个节点都有一个箭头指向后一个节点，首尾相连，组成一个圈。节点本身必须包含一个节点指针，用于指向后一个节点，除了这个节点指针是必须有的之外，节点本身还可以携带一些私有信息。



2. 双向链表

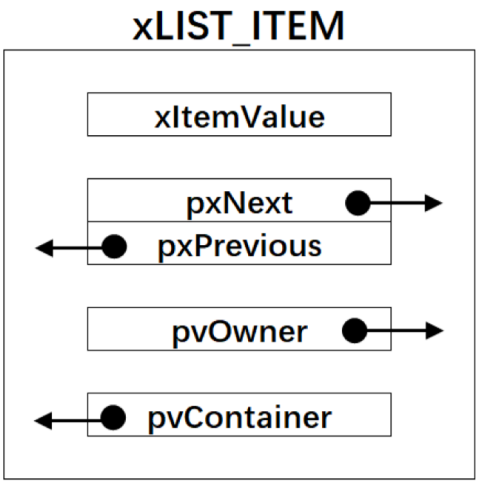
   双向链表与单向链表的区别就是节点中有两个节点指针，分别指向前后两个节点，其他完全一样。



B. FreeRTOS中的链表实现与操作

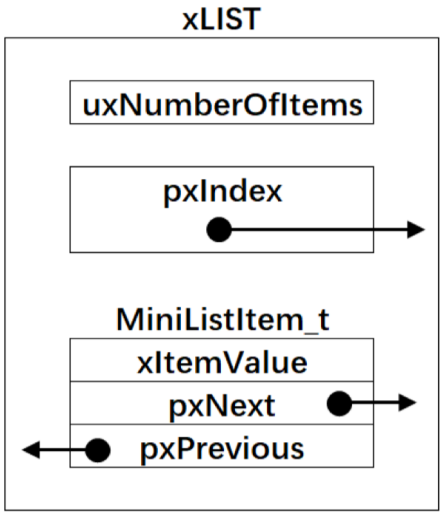
1. 链表（list）与节点（list item）

   - **链表节点**



```
struct xLIST_ITEM {
  TickType_t xItemValue;                    /* 辅助值，用于帮助节点做顺序排列 */
  struct xLIST_ITEM * pxNext;               /* 指向链表下一个节点 */
  struct xLIST_ITEM * pxPrevious;           /* 指向链表前一个节点 */
  void * pvOwner;                           /* 指向拥有该节点的内核对象，通常是 TCB */
  struct xLIST * configLIST_VOLATILE pxContainer;    /* 指向该节点所在的链表 */
};
```

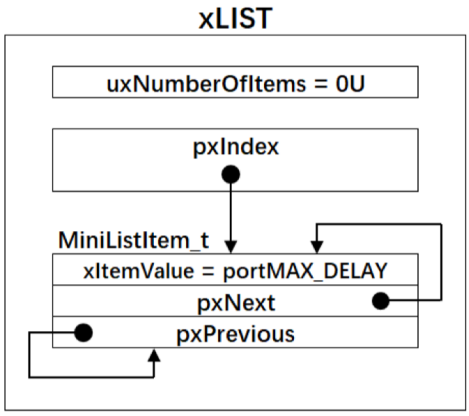
- **链表**




```
typedef struct xLIST {
  UBaseType_t uxNumberOfItems;  /* 链表节点计数器 */
  ListItem_t * pxIndex;         /* 链表节点索引指针 */
  MiniListItem_t xListEnd;      /* 链表最后一个节点*/
} List_t;
```

2. 链表初始化

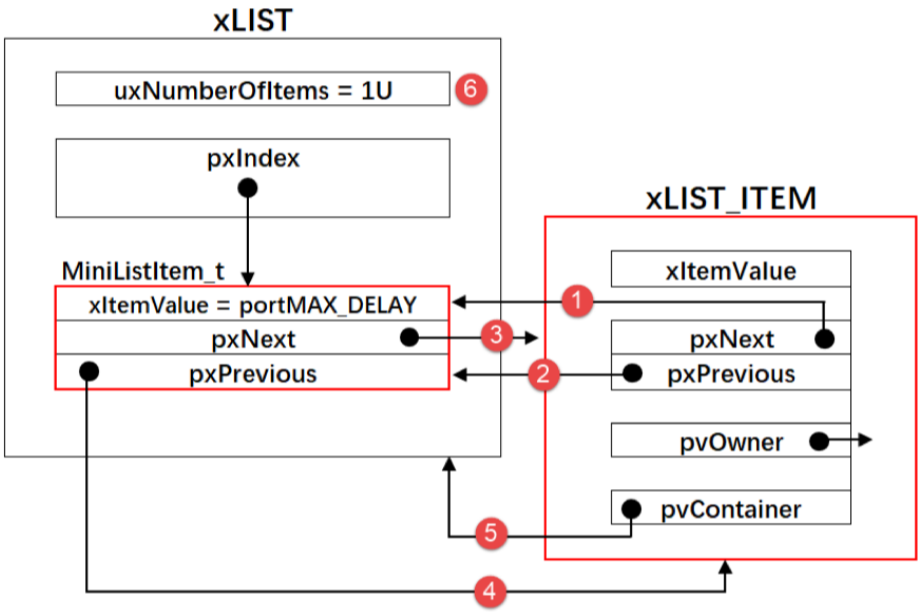
- **链表根节点初始化**



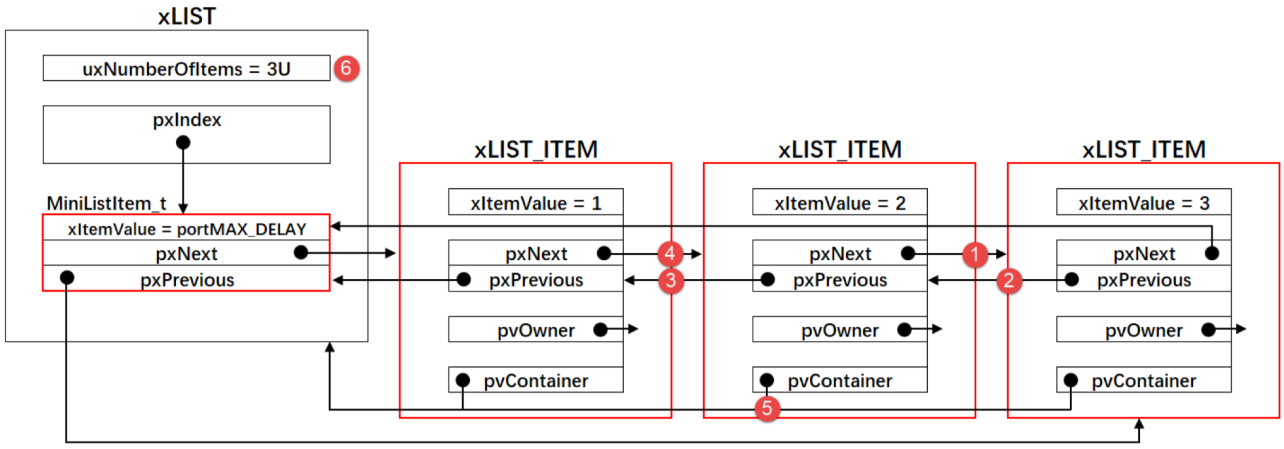
```
void vListInitialise( List_t * const pxList ){
  /*将链表索引指针指向最后一个节点，即第一个节点，或者第零个节点更准确，因为这个节点不会算入节点计数器的值*/
  pxList->pxIndex = ( ListItem_t * ) &( pxList->xListEnd );
  /*将链表最后（也可以理解为第一）一个节点的辅助排序的值设置为最大，确保该节点就是链表的最后节点（也可以理解为第一）*/
  pxList->xListEnd.xItemValue = portMAX_DELAY;
  /*将最后一个节点（也可以理解为第一）的 pxNext 和 pxPrevious 指针均指向节点自身，表示链表为空 */
  pxList->xListEnd.pxNext = ( ListItem_t * ) &( pxList->xListEnd );
  pxList->xListEnd.pxPrevious = ( ListItem_t * ) &( pxList->xListEnd );
  /*初始化链表节点计数器的值为 0，表示链表为空*/
  pxList->uxNumberOfItems = ( UBaseType_t ) 0U;
}
```

3. 节点的插入语删除操作

   - **将节点插入到链表的尾部** `void vListInsertEnd( List_t * const pxList, ListItem_t * const pxNewListItem )`



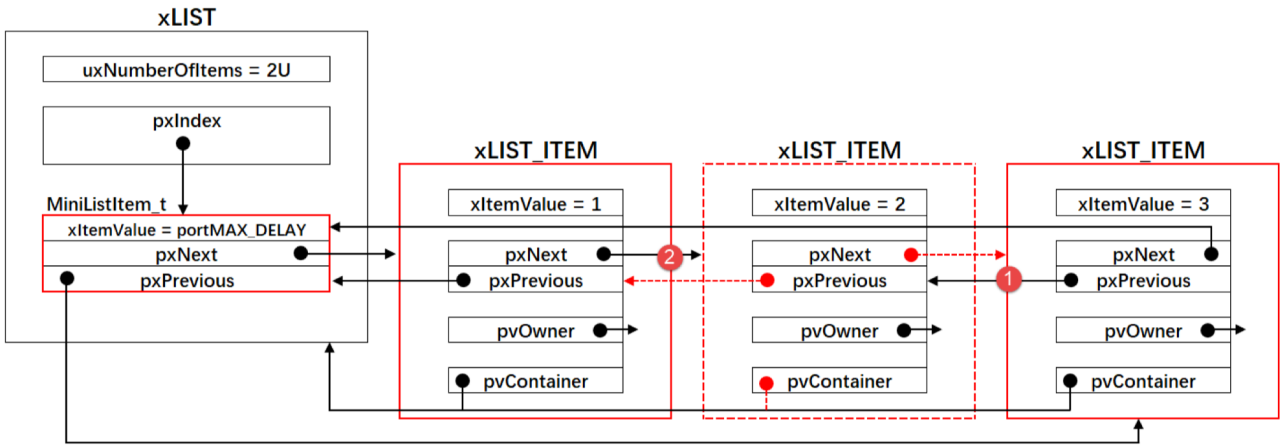
   - **将节点按照升序排列插入到链表** `void vListInsert( List_t * const pxList, ListItem_t * const pxNewListItem )`




```
void vListInsert( List_t * const pxList, ListItem_t * const pxNewListItem ){
  ListItem_t *pxIterator;
```

```
  const TickType_t xValueOfInsertion = pxNewListItem->xItemValue;

  /* Insert the new list item into the list, sorted in xItemValue order.
  If the list already contains a list item with the same item value then the
  new list item should be placed after it.  This ensures that TCBs which are
  stored in ready lists (all of which have the same xItemValue value) get a
  share of the CPU.  However, if the xItemValue is the same as the back marker
  the iteration loop below will not end.  Therefore the value is checked
  first, and the algorithm slightly modified if necessary. */
  if( xValueOfInsertion == portMAX_DELAY ){
    pxIterator = pxList->xListEnd.pxPrevious;
  }else{
    for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->pxNext->xItemValue <= xValueOfInsertion; pxIterator = pxIterator->pxNext )
  }

  pxNewListItem->pxNext = pxIterator->pxNext;
  pxNewListItem->pxNext->pxPrevious = pxNewListItem;
  pxNewListItem->pxPrevious = pxIterator;
  pxIterator->pxNext = pxNewListItem;

  /* Remember which list the item is in.  This allows fast removal of the
  item later. */
  pxNewListItem->pxContainer = pxList;

  ( pxList->uxNumberOfItems )++;
}
```

- 将节点从链表删除 `UBaseType_t uxListRemove( ListItem_t * const pxItemToRemove )`



C. FreeRTOS中使用的链表

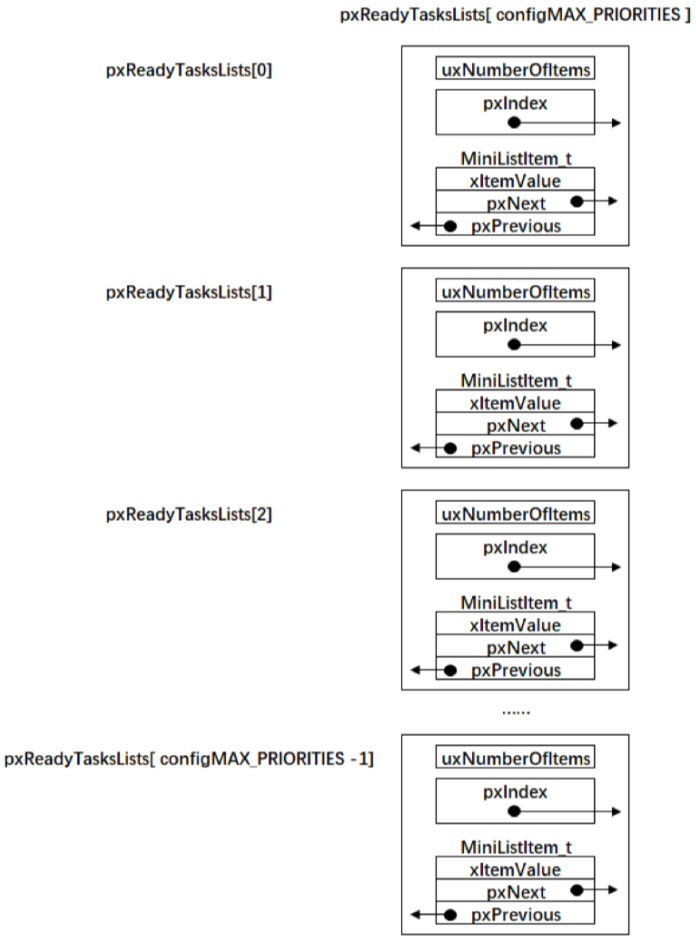
- `List_t pxReadyTasksLists[configMAX_PRIORITIES]`: Prioritised ready tasks.
- `List_t xDelayedTaskList1`: Delayed tasks for delays.
- `List_t xDelayedTaskList2`: Delayed tasks for overflowed the current tick count.
- `List_t xPendingReadyList`: Tasks that have been readied while the scheduler was suspended. They will be moved to the ready list when the scheduler is resumed.
- `List_t xTasksWaitingTermination`: Tasks that have been deleted - but their memory not yet been freed.
- `List_t xSuspendedTaskList`: Tasks that are currently suspended.

1. **就绪列表** `pxReadyTasksLists[configMAX_PRIORITIES]`

   就绪列表定义：

   任务创建好之后，需要把任务添加到就绪列表里面，表示任务已经就绪，系统随时可以调度。

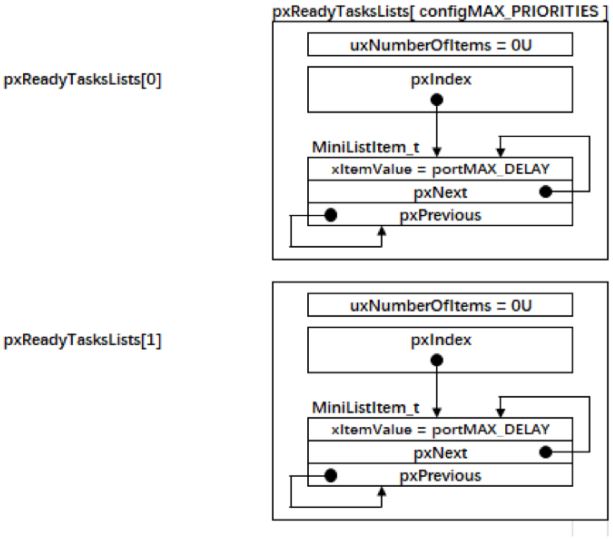
   就绪列表实际上就是一个`List_t`类型的数组，数组的大小由决定最大任务优先级的宏`configMAX_PRIORITIES`决定，`configMAX_PRIORITIES`在`FreeRTOSConfig.h`中默认定义为5，最大支持256个优先级。数组的下标对应了任务的优先级，同一优先级的任务统一插入到就绪列表的同一条链表中。



   就序列表初始化 `void prvInitialiseTaskLists( void )`：

```
static void prvInitialiseTaskLists( void ){
  UBaseType_t uxPriority;

  for( uxPriority = ( UBaseType_t ) 0U; uxPriority < ( UBaseType_t )
configMAX_PRIORITIES; uxPriority++ ){
    vListInitialise( &( pxReadyTasksLists[ uxPriority ] ) );
  }

  vListInitialise( &xDelayedTaskList1 );
  vListInitialise( &xDelayedTaskList2 );
  vListInitialise( &xPendingReadyList );

  #if ( INCLUDE_vTaskDelete == 1 ){
    vListInitialise( &xTasksWaitingTermination );
  }
  #endif /* INCLUDE_vTaskDelete */

  #if ( INCLUDE_vTaskSuspend == 1 ){
    vListInitialise( &xSuspendedTaskList );
```

```
  }
  #endif /* INCLUDE_vTaskSuspend */

  /* Start with pxDelayedTaskList using list1 and the pxOverflowDelayedTaskList
  using list2. */
  pxDelayedTaskList = &xDelayedTaskList1;
  pxOverflowDelayedTaskList = &xDelayedTaskList2;
}
```

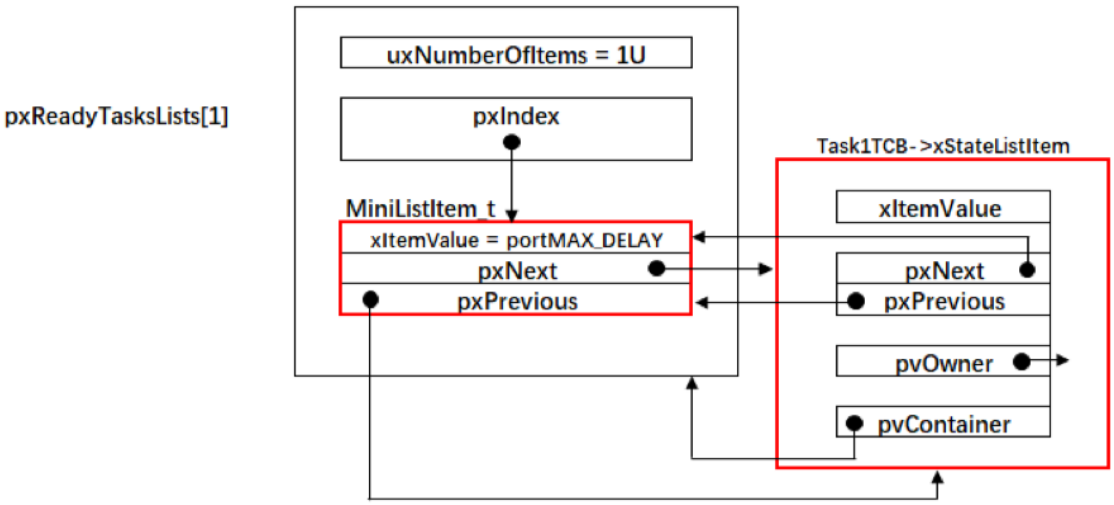


将任务插入到就绪列表 `void prvAddNewTaskToReadyList( TCB_t *pxNewTCB )`:

任务控制块里面有一个`xStateListItem`成员，数据类型为`ListItem_t`，将任务插入到就绪列表里面，就是通过将任务控制块的`xStateListItem`这个节点插入到就绪列表中来实现的。

```
static void prvAddNewTaskToReadyList( TCB_t *pxNewTCB ){
  /* Ensure interrupts don't access the task lists while the lists are being
updated. */
  taskENTER_CRITICAL();
  {
    uxCurrentNumberOfTasks++;
    if( pxCurrentTCB == NULL ){
      /* There are no other tasks, or all the other tasks are in the suspended state
- make this the current task. */
      pxCurrentTCB = pxNewTCB;
      if( uxCurrentNumberOfTasks == ( UBaseType_t ) 1 ){
        /* This is the first task to be created so do the preliminary initialisation
required.  We will not recover if this call fails, but we will report the failure.
*/
        prvInitialiseTaskLists();
      }
    }else{
      /* If the scheduler is not already running, make this task the current task if
it is the highest priority task to be created so far. */
      if( xSchedulerRunning == pdFALSE ){
        if( pxCurrentTCB->uxPriority <= pxNewTCB->uxPriority ){
          pxCurrentTCB = pxNewTCB;
        }
      }
    }
```

```
    uxTaskNumber++;

    /* Place the task represented by pxTCB into the appropriate ready list for the
task.  It is inserted at the end of the list.*/
    prvAddTaskToReadyList( pxNewTCB );
  }
  taskEXIT_CRITICAL();

  if( xSchedulerRunning != pdFALSE ){
    /* If the created task is of a higher priority than the current task then it
should run now. */
    if( pxCurrentTCB->uxPriority < pxNewTCB->uxPriority ){
      taskYIELD_IF_USING_PREEMPTION();
    }
  }
}
```



II. 任务管理

系统为了顺利的调度任务，为每个任务都额外定义了一个任务控制块，这个任务控制块就相当于任务的身份证，里面存有任务的所有信息，比如任务的栈指针、任务名称、任务的形参等。 有了这个任务控制块之后，以后系统对任务的全部操作都可以通过这个**任务控制块(TCB)**来实现。

```
/*
 * Task control block.  A task control block (TCB) is allocated for each task,
 * and stores task state information, including a pointer to the task's context
 * (the task's run time environment, including register values)
 */
typedef struct tskTaskControlBlock {
  StackType_t   *pxTopOfStack;    /*栈顶*/
  ListItem_t    xStateListItem;   /*任务节点*/
  ListItem_t    xEventListItem;   /*< Used to reference a task from an event list. */
  UBaseType_t   uxPriority;       /*优先级*/
  StackType_t   *pxStack;         /*当前栈帧*/
  char          pcTaskName[ configMAX_TASK_NAME_LEN ];

  #if ( ( portSTACK_GROWTH > 0 ) || ( configRECORD_STACK_HIGH_ADDRESS == 1 ) )
  StackType_t   *pxEndOfStack;
  #endif
```

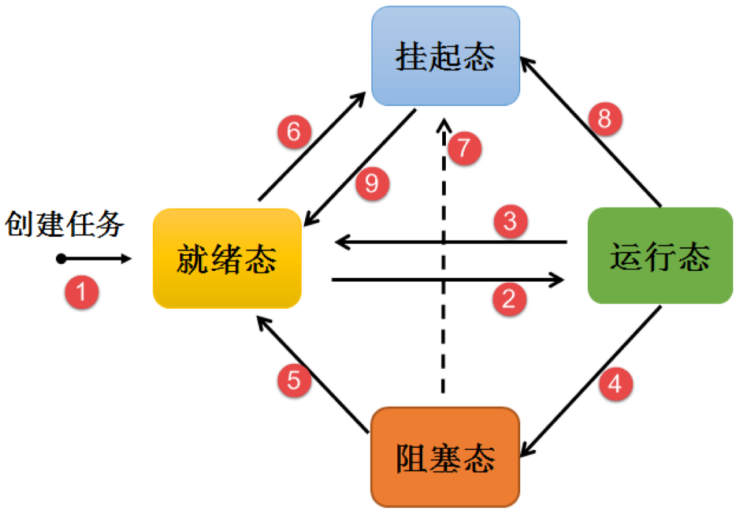
```
  #if ( portCRITICAL_NESTING_IN_TCB == 1 )
  UBaseType_t   uxCriticalNesting;  /*< Holds the critical section nesting depth for
ports that do not maintain their own count in the port layer. */
  #endif

  #if ( configUSE_MUTEXES == 1 )
  UBaseType_t   uxBasePriority;   /*< The priority last assigned to the task - used by
the priority inheritance mechanism. */
  UBaseType_t   uxMutexesHeld;
  #endif
} tskTCB;
```

A. 任务状态及转化

FreeRTOS任务状态通常分为以下四种：

- 就绪（Ready）：该任务在就绪列表中，就绪的任务已经具备执行的能力，只等待调度器进行调度，新创建的任务会初始化为就绪态。
- 运行（Running）：该状态表明任务正在执行，此时它占用处理器，FreeRTOS 调度器选择运行的永远是处于最高优先级的就绪态任务，当任务被运行的一刻，它的任务状态就变成了运行态。
- 阻塞（Blocked）：如果任务当前正在等待某个时序或外部中断，我们就说这个任务处于阻塞状态，该任务不在就绪列表中。包含任务被挂起、任务被延时、任务正在等待信号量、读写队列或者等待读写事件等。
- 挂起态（Suspended）：处于挂起态的任务对调度器而言是不可见的，让一个任务进入挂起状态的唯一办法就是调用 `vTaskSuspend()` 函数；而把一个挂起状态的任务恢复的唯一途径就 是调用 `vTaskResume()` 或 `vTaskResumeFromISR()` 函数

> 挂起态与阻塞态的区别，当任务有较长的时间不允许运行的时候，可以挂起任务，这样子调度器就不会管这个任务的任何信息，直到我们调用恢复任务的 API 函数；而任务处于阻塞态的时候，系统还需要判断阻塞态的任务是否超时，是否可以解除阻塞。



1. 创建任务➡️就绪态（Ready）：任务创建完成后进入就绪态，表明任务已准备就绪， 随时可以运行，只等待调度器进行调度
2. 就绪态➡️运行态（Running）：发生任务切换时，就绪列表中最高优先级的任务被执 行，从而进入运行态
3. 运行态➡️就绪态：有更高优先级任务创建或者恢复后，会发生任务调度，此刻就绪 列表中最高优先级任务变为运行态，那么原先运行的任务由运行态变为就绪态，依然在就绪列表中，等待最高优先级的任务运行完毕继续运行原来的任务
4. 运行态➡️阻塞态（Blocked）：正在运行的任务发生阻塞（挂起、延时、读信号量等）时，该任务会从就绪列表中删除，任务状态由运行态变成阻塞态，然后发生任务切换，运行就绪列表中当前最高优先级任务

5. 阻塞态➡️就绪态：阻塞的任务被恢复后（任务恢复、延时时间超时、读信号量超时或读到信号量等），此时被恢复的任务会被加入就绪列表，从而由阻塞态变成就绪态；如果此时 被恢复任务的优先级高于正在运行任务的优先级，则会发生任务切换，将该任务将再次转换任务 状态，由就绪态变成运行态
6. 就绪态➡️挂起态（Suspended）：任务可以通过调用 vTaskSuspend() API 函数都可以将处于任何状态的任务挂起，被挂起的任务得不到 CPU 的使用权，也不会参与调度，除非它从挂起态中解除
7. 阻塞态➡️挂起态（Suspended）：任务可以通过调用 vTaskSuspend() API 函数都可以将处于任何状态的任务挂起，被挂起的任务得不到 CPU 的使用权，也不会参与调度，除非它从挂起态中解除
8. 运行态➡️挂起态（Suspended）：任务可以通过调用 vTaskSuspend() API 函数都可以将处于任何状态的任务挂起，被挂起的任务得不到 CPU 的使用权，也不会参与调度，除非它从挂起态中解除
9. 挂起态➡️就绪态：把一个挂起状态的任务恢复的唯一途径就是调用 vTaskResume() 或 vTaskResumeFromISR() API 函数，如果此时被恢复任务的优先级高于正在运行任务的优先级， 则会发生任务切换，将该任务将再次转换任务状态，由就绪态变成运行态

1. **创建任务**：xTaskCreate (...)

FreeRTOS提供的关于任务创建的API：

- TaskHandle_t xTaskCreateStatic(...)：用户分配任务堆栈内存和TCB内存，称为静态任务
- BaseType_t xTaskCreateRestrictedStatic(...)：MPU保护下的静态任务创建
- BaseType_t xTaskCreateRestricted(...)：MPU保护下的动态任务创建
- BaseType_t xTaskCreate(...)：动态任务创建，系统从系统堆栈中分配堆栈内存和TCB内存

  其调用的函数包括：

  - pvPortMalloc(...)
  - prvInitialiseNewTask(...)：初始化任务堆栈和TCB，调用prvTaskExitError(void)

  - prvAddNewTaskToReadyList(...)：之前已介绍。

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
            const char * const pcName,
            const configSTACK_DEPTH_TYPE usStackDepth,
            void * const pvParameters,
            UBaseType_t uxPriority,
            TaskHandle_t * const pxCreatedTask ){
  TCB_t *pxNewTCB;
  BaseType_t xReturn;
```

```
    /* Allocate space for the TCB.  Where the memory comes from depends on
    the implementation of the port malloc function and whether or not static
    allocation is being used. */
    pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );
    if( pxNewTCB != NULL ) {
      /* Allocate space for the stack used by the task being created.
      The base of the stack memory stored in the TCB so the task can
      be deleted later if required. */
      pxNewTCB->pxStack = (StackType_t*)pvPortMalloc((((size_t)usStackDepth) *
sizeof(StackType_t)));
      if( pxNewTCB->pxStack == NULL ){
        /* Could not allocate the stack.  Delete the allocated TCB. */
        vPortFree( pxNewTCB );
        pxNewTCB = NULL;
      }
    }
  if( pxNewTCB != NULL ){
    prvInitialiseNewTask( pxTaskCode, pcName, ( uint32_t ) usStackDepth,
pvParameters, uxPriority, pxCreatedTask, pxNewTCB, NULL );
    prvAddNewTaskToReadyList( pxNewTCB );
    xReturn = pdPASS;
  }else{
    xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
  }
  return xReturn;
}
```

2. **删除任务**：`vTaskDelete( ... )`

> `vTaskDelete()`用于删除一个任务。当一个任务删除另外一个任务时，形参为要删除任务创建时返回的任务句柄，如果是删除自身，则形参为`NULL`。要想使用该函数必须在FreeRTOSConfig.h中把`INCLUDE_vTaskDelete`定义为1，删除的任务将从所有就绪、阻塞、挂起和事件列表中删除。
>
> - 将任务从就绪列表中删除，如果删除后就绪列表的长度为0，当前没有就绪的任务，应该调用`taskRESET_READY_PRIORITY()`函数清除任务的最高就绪优先级变量 `uxTopReadyPriority`中的位
> - 如果当前任务在等待事件，那么将任务从事件列表中移除
> - 如果此时删除的任务是任务自身的话，那么删除任务函数不能在任务本身内完成，因为需要上下文切换到另一个任务。所以需要将任务放在结束列表中 `xTasksWaitingTermination`，空闲任务会检查结束列表并在空闲任务中释放删除任务的控制块和已删除任务的堆栈内存
> - 增加`uxDeletedTasksWaitingCleanUp`变量的值，该变量用于记录有多少个任务需要释放内存，以便空闲任务知道有多少个已删除的任务需要进行内存释放，空闲任务会检查结束列表`xTasksWaitingTermination`并且释放对应删除任务的内存空间，空闲任务调用`prvCheckTasksWaitingTermination()`函数进行这些相应操作，该函数是FreeRTOS内部调用的函数，在`prvIdleTask`中调用

```
void vTaskDelete( TaskHandle_t xTaskToDelete ){
  TCB_t *pxTCB;
  taskENTER_CRITICAL();
  {
    /* If null is passed in here then it is the calling task that is being deleted.
*/
    pxTCB = prvGetTCBFromHandle( xTaskToDelete );
```

```
    /* Remove task from the ready list. */
    if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( UBaseType_t ) 0 ){
      taskRESET_READY_PRIORITY( pxTCB->uxPriority );
    }

    /* Is the task waiting on an event also? */
    if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) != NULL ){
      ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
    }

    /* Increment the uxTaskNumber also so kernel aware debuggers can detect that the
task lists need re-generating.  This is done before
    portPRE_TASK_DELETE_HOOK() as in the Windows port that macro will not return. */
    uxTaskNumber++;

    if( pxTCB == pxCurrentTCB ){
      /* A task is deleting itself.  This cannot complete within the task itself, as
a context switch to another task is required.
      Place the task in the termination list.  The idle task will check the
termination list and free up any memory allocated by
      the scheduler for the TCB and stack of the deleted task. */
      vListInsertEnd( &xTasksWaitingTermination, &( pxTCB->xStateListItem ) );

      /* Increment the ucTasksDeleted variable so the idle task knows there is a
task that has been deleted and that it should therefore
      check the xTasksWaitingTermination list. */
      ++uxDeletedTasksWaitingCleanUp;

      /* The pre-delete hook is primarily for the Windows simulator, in which
Windows specific clean up operations are performed,
      after which it is not possible to yield away from this task - hence
xYieldPending is used to latch that a context switch is
      required. */
      portPRE_TASK_DELETE_HOOK( pxTCB, &xYieldPending );
    }else{
      --uxCurrentNumberOfTasks;
      prvDeleteTCB( pxTCB );

      /* Reset the next expected unblock time in case it referred to the task that
has just been deleted. */
      prvResetNextTaskUnblockTime();
    }
  }
  taskEXIT_CRITICAL();

  /* Force a reschedule if it is the currently running task that has just been
deleted. */
  if( xSchedulerRunning != pdFALSE ){
    if( pxTCB == pxCurrentTCB ){
      portYIELD_WITHIN_API();
    }
  }
}
```

3. **任务时延1**：vTaskDelay(...)

vTaskDelay()用于阻塞延时，调用该函数后，**任务将进入阻塞状态**，进入阻塞态的任务将让出CPU资源。延时的时长由形参xTicksToDelay决定，单位为系统节拍周期，比如系统的时钟节拍周期为1ms，那么调用`vTaskDelay(100)的延时时间则为100ms。

vTaskDelay()延时是**相对性的延时**，它指定的延时时间是从调用vTaskDelay()结束后开始计算的，经过指定的时间后延时结束。比如vTaskDelay(100)，从调用vTaskDelay()结束后，任务进入阻塞状态，经过100个系统时钟节拍周期后，任务解除阻塞。因此，vTaskDelay()并不适用与周期性执行任务的场合。此外，其它任务和中断活动，也会影响到vTaskDelay()的调用（比如调用前高优先级任务抢占了当前任务），进而影响到任务的下一次执行的时间。

具体实现逻辑如下：

- 延时时间 xTicksToDelay 要大于 0 个 tick，否则会进行强制切换任务
- 挂起任务调度器
- 将任务添加到延时列表中
- 恢复任务调度器
- 强制切换任务，调用portYIELD_WITHIN_API()函数将PendSV的bit28置1

```
void vTaskDelay( const TickType_t xTicksToDelay ){
  BaseType_t xAlreadyYielded = pdFALSE;

  /* A delay time of zero just forces a reschedule. */
  if( xTicksToDelay > ( TickType_t ) 0U ){
    vTaskSuspendAll();
    {
      /* A task that is removed from the event list while the scheduler is suspended
will not get placed in the ready
      list or removed from the blocked list until the scheduler is resumed.

      This task cannot be in an event list as it is the currently executing task. */
      prvAddCurrentTaskToDelayedList( xTicksToDelay, pdFALSE );
    }
    xAlreadyYielded = xTaskResumeAll();
  }

  /* Force a reschedule if xTaskResumeAll has not already done so, we may have put
ourselves to sleep. */
  if( xAlreadyYielded == pdFALSE ){
    portYIELD_WITHIN_API();
  }
}
```

具体实现逻辑如下：

- xCanBlockIndefinitely表示是否可以永久阻塞，如果pdFALSE表示不允许永久阻塞，也就是不允许挂起当然任务，而如果是pdTRUE，则可以永久阻塞
- 获取当前调用延时函数的时间点
- 在将任务添加到阻塞列表之前，从就绪列表中删除任务，因为两个列表都使用相同的列表项。调用uxListRemove()函数将任务从就绪列表中删除
- 在支持任务挂起时，如果xTicksToWait == portMAX_DELAY，则将当前任务挂起，此操作必须将INCLUDE_vTaskSuspend宏定义使能，并且xCanBlockIndefinitely为 pdTRUE时，调用vListInsertEnd()函数直接将任务添加到挂起列表xSuspendedTaskList，而不是延时列表
- 计算唤醒任务的时间
- 唤醒时间如果溢出了，则会将任务添加到延时溢出列表中，任务的延时由两个列表来维护，一个是用于延时溢出情况，另一个用于非溢出情况

- 如果唤醒任务的时间没有溢出，就会将任务添加到延时列表中，而不是延时溢出列表
- 如果下一个要唤醒的任务就是当前延时的任务，那么就需要重置下一个任务的解除阻塞时间 `xNextTaskUnblockTime` 为唤醒当前延时任务的时间 `xTimeToWake`

```
static void prvAddCurrentTaskToDelayedList( TickType_t xTicksToWait, const
BaseType_t xCanBlockIndefinitely ){
  TickType_t xTimeToWake;
  const TickType_t xConstTickCount = xTickCount;

  /* Remove the task from the ready list before adding it to the blocked list as the
same list item is used for both lists. */
  if( uxListRemove( &( pxCurrentTCB->xStateListItem ) ) == ( UBaseType_t ) 0 ){
    /* The current task must be in a ready list, so there is no need to check, and
the port reset macro can be called directly. */
    portRESET_READY_PRIORITY( pxCurrentTCB->uxPriority, uxTopReadyPriority );
  }

    if( ( xTicksToWait == portMAX_DELAY ) && ( xCanBlockIndefinitely != pdFALSE ) )
    {
      /* Add the task to the suspended task list instead of a delayed task list to
ensure it is not woken by a timing event.  It will block
      indefinitely. */
      vListInsertEnd( &xSuspendedTaskList, &( pxCurrentTCB->xStateListItem ) );
    }else{
      /* Calculate the time at which the task should be woken if the event does not
occur.  This may overflow but this doesn't matter, the
      kernel will manage it correctly. */
      xTimeToWake = xConstTickCount + xTicksToWait;

      /* The list item will be inserted in wake time order. */
      listSET_LIST_ITEM_VALUE( &( pxCurrentTCB->xStateListItem ), xTimeToWake );

      if( xTimeToWake < xConstTickCount ){
        /* Wake time has overflowed.  Place this item in the overflow list. */
        vListInsert( pxOverflowDelayedTaskList, &( pxCurrentTCB->xStateListItem ) );
      }else{
        /* The wake time has not overflowed, so the current block list is used. */
        vListInsert( pxDelayedTaskList, &( pxCurrentTCB->xStateListItem ) );

        /* If the task entering the blocked state was placed at the head of the list
of blocked tasks then xNextTaskUnblockTime
        needs to be updated too. */
        if( xTimeToWake < xNextTaskUnblockTime ){
          xNextTaskUnblockTime = xTimeToWake;
        }
      }
    }
  }
}
```

4. **任务时延2**：`vTaskDelayUntil( ... )`

`vTaskDelayUntil()`与`vTaskDelay()`一样都是用来实现任务的周期性延时。但`vTaskDelay()`的延时是相对的，是不确定的，它的延时是等`vTaskDelay()`调用完毕后开始计算的。并且`vTaskDelay()`延时的时间到了之后，如果有高优先级的任务或者中断正在执行，被延时阻塞的任务并不会马上解除阻塞，所有每次执行任务的周期并不完全确定。而`vTaskDelayUntil()`延时是绝对的，适用于周期性执行的任务。
当`(*pxPreviousWakeTime + xTimeIncrement)`时间到达后，`vTaskDelayUntil()`函数立刻返回，如果任务是最高优先级的，那么任务会立马解除阻塞。

- `TickType_t * const pxPreviousWakeTime`指向一个变量，该变量保存任务最后一次解除阻塞的的时刻。第一次使用时，该变量必须初始化为当前时间，之后这个变量会在`vTaskDelayUntil()`函数内自动更新
  - xTimeIncrement：任务周期时间
  - pxPreviousWakeTime：上一次唤醒任务的时间点
  - xTimeToWake：本次要唤醒任务的时间点
  - xConstTickCount：进入延时的时间点
- `xTimeToWake`周期循环时间，当时间等于`(*pxPreviousWakeTime + xTimeIncrement)`时，任务解除阻塞。如果不改变参数`xTimeIncrement`的值，调用该函数的任务会按照固定频率执行
- 获取开始进行延时的时间点
- 计算延时到达的时间，也就是唤醒任务的时间，由于变量`xTickCount`与`xTimeToWake`可能会溢出，所以程序必须检测各种溢出情况
- `pxPreviousWakeTime`中保存的是上次唤醒时间，唤醒后需要一定时间执行任务主体代码，如果上次唤醒时间大于当前时间，说明节拍计数器溢出了
- 如果本次任务的唤醒时间小于上次唤醒时间，但是大于开始进入延时的时间，进入延时的时间与任务唤醒时间都已经溢出了，这样就可以看做没有溢出
- 更新上一次唤醒任务的时间`pxPreviousWakeTime`
- `prvAddCurrentTaskToDelayedList()`函数需要的是阻塞时间而不是唤醒时间，因此减去当前的进入延时的时间`xConstTickCount`
- 强制执行一次上下文切换

```
void vTaskDelayUntil( TickType_t * const pxPreviousWakeTime, const TickType_t
xTimeIncrement ){
  TickType_t xTimeToWake;
  BaseType_t xAlreadyYielded, xShouldDelay = pdFALSE;

  vTaskSuspendAll();
  {
    /* Minor optimisation.  The tick count cannot change in this block. */
    const TickType_t xConstTickCount = xTickCount;

    /* Generate the tick time at which the task wants to wake. */
    xTimeToWake = *pxPreviousWakeTime + xTimeIncrement;

    if( xConstTickCount < *pxPreviousWakeTime ){
      /* The tick count has overflowed since this function was lasted called.  In
this case the only time we should ever
      actually delay is if the wake time has also overflowed, and the wake time is
greater than the tick time.  When this
      is the case it is as if neither time had overflowed. */
      if( ( xTimeToWake < *pxPreviousWakeTime ) && ( xTimeToWake > xConstTickCount )
){
```

```
        xShouldDelay = pdTRUE;
      }
    }else{
      /* The tick time has not overflowed.  In this case we will delay if either the
wake time has overflowed, and/or the
      tick time is less than the wake time. */
      if( ( xTimeToWake < *pxPreviousWakeTime ) || ( xTimeToWake > xConstTickCount )
){
        xShouldDelay = pdTRUE;
      }
    }

    /* Update the wake time ready for the next call. */
    *pxPreviousWakeTime = xTimeToWake;

    if( xShouldDelay != pdFALSE ){
      /* prvAddCurrentTaskToDelayedList() needs the block time, not the time to
wake, so subtract the current tick count. */
      prvAddCurrentTaskToDelayedList( xTimeToWake - xConstTickCount, pdFALSE );
    }
  }
  xAlreadyYielded = xTaskResumeAll();

  /* Force a reschedule if xTaskResumeAll has not already done so, we may have put
ourselves to sleep. */
  if( xAlreadyYielded == pdFALSE ){
    portYIELD_WITHIN_API();
  }
}
```

5. **挂起任务**：`vTaskSuspend( ... )`

> 挂起指定任务。被挂起的任务绝不会得到 CPU 的使用权，不管该任务具有什么优先级。任务可以通过调用 `vTaskSuspend()` 函数都可以将处于任何状态的任务挂起，被挂起的任务得不到CPU的使用权，也不会参与调度，它相对于调度器而言是不可见的，除非它从挂起态中解除。**此函数的主体逻辑如下：**
>
> - 从就绪/阻塞列表中删除即将要挂起的任务，然后更新最高优先级变量 `uxReadyPriorities`
> - 如果任务在等待事件，也将任务从等待事件列表中移除
> - 将任务状态添加到挂起列表中。在FreeRTOS中有专门的列表用于记录任务的状态，记录任务挂起态的列表就是 `xSuspendedTaskList`，所有被挂起的任务都会放到这个列表中
> - 重置下一个任务的解除阻塞时间。重新计算一下还要多长时间执行下一个任务，如果下个任务的解锁，刚好是被挂起的那个任务，那么就是不正确的了，因为挂起的任务对调度器而言是不可见的，所以调度器是无法对挂起态的任务进行调度，所以要重新从延时列表中获取下一个要解除阻塞的任务
> - 如果挂起的是当前运行中的任务，并且调度器已经是运行的，则需要立即切换任务。不然系统的任务就错乱了，这是不允许的
> - 调度器未运行（`xSchedulerRunning == pdFALSE`），但 `pxCurrentTCB` 指向的任务刚刚被挂起，所以必须重置 `pxCurrentTCB` 以指向其他可以运行的任务
> - 首先调用函数 `listCURRENT_LIST_LENGTH()` 判断一下系统中所有的任务是不是都被挂起了，也就是查看列表 `xSuspendedTaskList` 的长度是不是等于 `uxCurrentNumberOfTasks`， 事实上并不会发生这种情况，因为空闲任务是不允许被挂起和阻塞的，必须保证系统中无论如何都有一个任务可以运行
> - 如果没有其他任务准备就绪，因此将 `pxCurrentTCB` 设置为 `NULL`，在创建下一个任务时 `pxCurrentTCB` 将重新被设置。但是实际上并不会执行到这里，因为系统中的空闲任务永远是可以运行的
> - 如果有其他可运行的任务，则切换到其他任务

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend )
{
  TCB_t *pxTCB;
  taskENTER_CRITICAL();
  {
    /* If null is passed in here then it is the running task that is being
suspended. */
    pxTCB = prvGetTCBFromHandle( xTaskToSuspend );
    /* Remove task from the ready/delayed list and place in the suspended list. */
    if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( UBaseType_t ) 0 ){
      taskRESET_READY_PRIORITY( pxTCB->uxPriority );
    }
    /* Is the task waiting on an event also? */
    if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) != NULL ){
      ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
    }
    /* Add task to suspended list */
    vListInsertEnd( &xSuspendedTaskList, &( pxTCB->xStateListItem ) );
  }
  taskEXIT_CRITICAL();

  if( xSchedulerRunning != pdFALSE ){
    /* Reset the next expected unblock time in case it referred to the task that is
now in the Suspended state. */
    taskENTER_CRITICAL();
    {
      prvResetNextTaskUnblockTime();
    }
    taskEXIT_CRITICAL();
  }

  if( pxTCB == pxCurrentTCB ){
    if( xSchedulerRunning != pdFALSE ){
      /* The current task has just been suspended. */
      portYIELD_WITHIN_API();
    }else{
      /* The scheduler is not running, but the task that was pointed
      to by pxCurrentTCB has just been suspended and pxCurrentTCB
      must be adjusted to point to a different task. */
      if( listCURRENT_LIST_LENGTH( &xSuspendedTaskList ) == uxCurrentNumberOfTasks )
{
        /* No other tasks are ready, so set pxCurrentTCB back to
        NULL so when the next task is created pxCurrentTCB will
        be set to point to it no matter what its relative priority
        is. */
        pxCurrentTCB = NULL;
      }else{
        vTaskSwitchContext();
      }
    }
  }
}
```

6. **挂起调度器**：vTaskSuspendAll()

这个函数将所有的任务都挂起，从而将调度器锁定，并且这个函数是可以进行嵌套的。调度器被挂起后则不能进行上下文切换，但是中断还是使能的。当调度器被挂起的时候，如果有中断需要进行上下文切换，那么这个中断将会被挂起，在调度器恢复之后才响应这个中断。调度器恢复可以调用 `xTaskResumeAll()` 函数，调用了多少次的 `vTaskSuspendAll()` 就要调用多少次 `xTaskResumeAll()` 进行恢复。

`uxSchedulerSuspended` 用于记录调度器是否被挂起，该变量默认初始值为 `pdFALSE`，表明调度器是没被挂起的，每调用一次 `vTaskSuspendAll()` 函数就将变量加一，用于记录调用了多少次 `vTaskSuspendAll()` 函数进行恢复。

```
void vTaskSuspendAll( void ){
  ++uxSchedulerSuspended;
}
```

7. **恢复任务**：`vTaskResume( ... )`

任务恢复就是让挂起的任务重新进入就绪状态，恢复的任务会保留挂起前的状态信息，在恢复的时候根据挂起时的状态继续运行。如果被恢复任务在所有就绪态任务中，处于最高优先级列表的第一位，那么系统将进行任务上下文的切换。具体逻辑如下：

- `xTaskToResume` 是恢复指定任务的任务句柄，根据 `xTaskToResume` 任务句柄获取对应的任务控制块
- `pxTCB` 任务控制块指针不能为 `NULL`，肯定要已经挂起的任务才需要恢复，同时要恢复的任务不能是当前正在运行的任务，因为当前正在运行（运行态）的任务不需要恢复，只能恢复处于挂起态的任务
- 进入临界区，防止被打断
- 判断要恢复的任务是否真的被挂起了，如果被挂起才需要恢复，没被挂起那当然也不需要恢复
- 将要恢复的任务从挂起列表中删除。在FreeRTOS中有专门的列表用于记录任务的状态，记录任务挂起态的列表就是 `xSuspendedTaskList`，现在恢复任务就将要恢复的任务从列表中删除
- 将要恢复的任务添加到就绪列表中去，任务从挂起态恢复为就绪态。FreeRTOS也是有专门的列表记录处于就绪态的任务，这个列表就是 `pxReadyTasksLists`
- 如果恢复的任务优先级比当前正在运行的任务优先级更高，则需要进行任务的切换，调用 `taskYIELD_IF_USING_PREEMPTION()` 进行一次任务切换

```
void vTaskResume( TaskHandle_t xTaskToResume ){
  TCB_t * const pxTCB = xTaskToResume;

  /* The parameter cannot be NULL as it is impossible to resume the currently
executing task. */
  if( ( pxTCB != pxCurrentTCB ) && ( pxTCB != NULL ) ){
    taskENTER_CRITICAL();
    {
      if( prvTaskIsTaskSuspended( pxTCB ) != pdFALSE ){
        /* The ready list can be accessed even if the scheduler is suspended because
this is inside a critical section. */
        ( void ) uxListRemove(  &( pxTCB->xStateListItem ) );
        prvAddTaskToReadyList( pxTCB );

        /* A higher priority task may have just been resumed. */
        if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority ){
          /* This yield may not cause the task just resumed to run, but will leave
the lists in the correct state for the next yield. */
          taskYIELD_IF_USING_PREEMPTION();
        }
      }
    }
```

```
    taskEXIT_CRITICAL();
  }
}
```

8. **从ISR中恢复任务**：`xTaskResumeFromISR( ... )`

> `xTaskResumeFromISR()`与`vTaskResume()`一样都是用于恢复被挂起的任务，不一样的是`xTaskResumeFromISR()`专门用在中断服务程序中。无论通过调用一次或多次 `vTaskSuspend()`函数而被挂起的任务，也只需调用一次`xTaskResumeFromISR()`函数即可解挂。要想使用该函数必须在FreeRTOSConfig.h中把`INCLUDE_vTaskSuspend`和 `INCLUDE_vTaskResumeFromISR`都定义为1才有效。任务还没有处于挂起态的时候，调用`xTaskResumeFromISR()`函数是没有任何意义的。

```
BaseType_t xTaskResumeFromISR( TaskHandle_t xTaskToResume ){
  BaseType_t xYieldRequired = pdFALSE;
  TCB_t * const pxTCB = xTaskToResume;
  UBaseType_t uxSavedInterruptStatus;

  portASSERT_IF_INTERRUPT_PRIORITY_INVALID();

  uxSavedInterruptStatus = portSET_INTERRUPT_MASK_FROM_ISR();
  {
    if( prvTaskIsTaskSuspended( pxTCB ) != pdFALSE ){
      /* Check the ready lists can be accessed. */
      if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE ){
        /* Ready lists can be accessed so move the task from the suspended list to
the ready list directly. */
        if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority ){
          xYieldRequired = pdTRUE;
        }
        ( void ) uxListRemove( &( pxTCB->xStateListItem ) );
        prvAddTaskToReadyList( pxTCB );
      }else{
        /* The delayed/ready lists cannot be accessed so the task is held in the
pending ready list until the scheduler is unsuspended. */
        vListInsertEnd( &( xPendingReadyList ), &( pxTCB->xEventListItem ) );
      }
    }
  }
  portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSavedInterruptStatus );

  return xYieldRequired;
}
```

9. **恢复调度器**：`xTaskResumeAll()`

> 当调用了`vTaskSuspendAll()`函数将调度器挂起，想要恢复调度器的时候我们就需要调用`xTaskResumeAll()`函数。函数逻辑如下：
>
> - 如果调度器恢复正常工作，也就是调度器没有被挂起，就可以将所有待处理的就绪任务从待处理就绪列表`xPendingReadyList`移动到适当的就绪列表中
> - 当待处理就绪列表`xPendingReadyList`中是非空的时候，就需要将待处理就绪列表中的任务移除，添加到就绪列表中去

- 如果移动的任务的优先级高于当前任务，需要进行一次任务的切换，重置`xYieldPending = pdTRUE`表示需要进行任务切换
- 在调度器被挂起时，任务被解除阻塞，这可能阻止了重新计算下一个解除阻塞时间，在这种情况下，需要重置下一个任务的解除阻塞时间。调用 `prvResetNextTaskUnblockTime()`函数将从延时列表中获取下一个要解除阻塞的任务
- 如果在调度器挂起这段时间产生滴答定时器的计时，并且在这段时间有任务解除阻塞，由于调度器的挂起导致没法切换任务，当恢复调度器的时候应立即处理这些任务。这样既确保了滴答定时器的计数不会滑动，也保证了所有在延时的任务都会在正确的时间恢复
- 调用`xTaskIncrementTick()`函数查找是否有待进行切换的任务，如果有则应该进行任务切换
- 如果需要任务切换，则调用`taskYIELD_IF_USING_PREEMPTION()`函数发起一次任务切换

```
BaseType_t xTaskResumeAll( void ){
  TCB_t *pxTCB = NULL;
  BaseType_t xAlreadyYielded = pdFALSE;

  taskENTER_CRITICAL();
  {
    --uxSchedulerSuspended;

    if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE ){
      if( uxCurrentNumberOfTasks > ( UBaseType_t ) 0U ){
        /* Move any readied tasks from the pending list into the appropriate ready
list. */
        while( listLIST_IS_EMPTY( &xPendingReadyList ) == pdFALSE ){
          pxTCB = listGET_OWNER_OF_HEAD_ENTRY( ( &xPendingReadyList ) );
          ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
          ( void ) uxListRemove( &( pxTCB->xStateListItem ) );
          prvAddTaskToReadyList( pxTCB );

          /* If the moved task has a priority higher than the current task then a
yield must be performed. */
          if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority ){
            xYieldPending = pdTRUE;
          }
        }

        if( pxTCB != NULL ){
          /* A task was unblocked while the scheduler was suspended, which may have
prevented the next unblock time from being
          re-calculated, in which case re-calculate it now.  Mainly important for
low power tickless implementations, where
          this can prevent an unnecessary exit from low power state. */
          prvResetNextTaskUnblockTime();
        }

        /* If any ticks occurred while the scheduler was suspended then they should
be processed now.  This ensures the tick count does
        not slip, and that any delayed tasks are resumed at the correct time. */
        {
          UBaseType_t uxPendedCounts = uxPendedTicks; /* Non-volatile copy. */

          if( uxPendedCounts > ( UBaseType_t ) 0U ){
            do{
              if( xTaskIncrementTick() != pdFALSE ){
                xYieldPending = pdTRUE;
```

```
                }
                --uxPendedCounts;
              } while( uxPendedCounts > ( UBaseType_t ) 0U );
              uxPendedTicks = 0;
            }
          }

          if( xYieldPending != pdFALSE ){
              xAlreadyYielded = pdTRUE;
            }
            taskYIELD_IF_USING_PREEMPTION();
          }
        }
      }
    }
    taskEXIT_CRITICAL();
    return xAlreadyYielded;
}
```

III. 调度器管理

A. 启动调度器

> 调度器的启动由 `vTaskStartScheduler()` 函数来完成。其调用函数 `BaseType_t xPortStartScheduler( void )` 切换到任务执行。

```
void vTaskStartScheduler( void ){
    BaseType_t xReturn;

    /* Add the idle task at the lowest priority. */
    /* The Idle task is being created using dynamically allocated RAM. */
    xReturn = xTaskCreate(  prvIdleTask,
                configIDLE_TASK_NAME,
                configMINIMAL_STACK_SIZE,
                ( void * ) NULL,
                portPRIVILEGE_BIT, /* In effect ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), but tskIDLE_PRIORITY is zero. */
                &xIdleTaskHandle );

    if( xReturn == pdPASS ){
      xReturn = xTimerCreateTimerTask();
    }

    if( xReturn == pdPASS ){
    /* Interrupts are turned off here, to ensure a tick does not occur before or during the call to xPortStartScheduler().  The stacks of the created tasks contain a status word with interrupts switched on so interrupts will automatically get re-enabled when the first task starts to run. */
    portDISABLE_INTERRUPTS();
```

```
    xNextTaskUnblockTime = portMAX_DELAY;
    xSchedulerRunning = pdTRUE;
    xTickCount = ( TickType_t ) configINITIAL_TICK_COUNT;

    /* Setting up the timer tick is hardware specific and thus in the portable
interface. */
    if( xPortStartScheduler() != pdFALSE ){
      /* Should not reach here as if the scheduler is running the function will not
return. */
    }else{
      /* Should only reach here if a task calls xTaskEndScheduler(). */
    }
  }else{
    /* This line will only be reached if the kernel could not be started, because there
was not enough FreeRTOS heap to create the idle task or the timer task. */
    configASSERT( xReturn != errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY );
  }
  /* Prevent compiler warnings if INCLUDE_xTaskGetIdleTaskHandle is set to 0, meaning
xIdleTaskHandle is not used anywhere else. */
  ( void ) xIdleTaskHandle;
}
```

函数`BaseType_t xPortStartScheduler( void )`最终调用函数`void prvPortStartFirstTask( void )`切换到任务执行。

```
BaseType_t xPortStartScheduler( void ){
  /* Make PendSV and SysTick the lowest priority interrupts. */
  portNVIC_SYSPRI2_REG |= portNVIC_PENDSV_PRI;
  portNVIC_SYSPRI2_REG |= portNVIC_SYSTICK_PRI;

  /* Start the timer that generates the tick ISR.  Interrupts are disabled here already.
*/
  vPortSetupTimerInterrupt();

  /* Initialise the critical nesting count ready for the first task. */
  uxCriticalNesting = 0;

  /* Ensure the VFP is enabled - it should be anyway. */
  vPortEnableVFP();

  /* Lazy save always. */
  *( portFPCCR ) |= portASPEN_AND_LSPEN_BITS;

  /* Start the first task. */
  prvPortStartFirstTask();

  /* Should never get here as the tasks will now be executing!  Call the task
  exit error function to prevent compiler warnings about a static function
  not being called in the case that the application writer overrides this
  functionality by defining configTASK_RETURN_ADDRESS.  Call
  vTaskSwitchContext() so link time optimisation does not remove the
  symbol. */
  vTaskSwitchContext();
  prvTaskExitError();
```

```
  /* Should not get here! */
  return 0;
}
```

prvStartFirstTask()函数用于开始第一个任务，主要做了两个动作，一个是更新MSP的值，二是产生SVC系统调用，然后去到SVC的中断服务函数里面真正切换到第一个任务。

```
static void prvPortStartFirstTask( void ){
  /* Start the first task.  This also clears the bit that indicates the FPU is
  in use in case the FPU was used before the scheduler was started - which
  would otherwise result in the unnecessary leaving of space in the SVC stack
  for lazy saving of FPU registers. */
  __asm volatile(
          " ldr r0, =0xE000ED08   \n"   /* Use the NVIC offset register to locate the
stack. */
          " ldr r0, [r0]      \n"
          " ldr r0, [r0]      \n"
          " msr msp, r0     \n"         /* Set the msp back to the start of the stack.
*/
          " mov r0, #0      \n"         /* Clear the bit that indicates the FPU is in
use, see comment above. */
          " msr control, r0   \n"
          " cpsie i       \n"           /* Globally enable interrupts. */
          " cpsie f       \n"
          " dsb         \n"
          " isb         \n"
          " svc 0         \n"           /* NOTE: System call to start first task. */
          " nop         \n"
        );
}
```

SVC中断要想被成功响应，其函数名必须与向量表注册的名称一致，在启动文件的向量表中，SVC的中断服务函数注册的名称是SVC_Handler，所以SVC中断服务函数的名称我们应该写成 SVC_Handler，但是在FreeRTOS中，官方版本写的是vPortSVCHandler()，为了能够顺利的响应SVC中断，需要改中断向量表中SVC的注册的函数名称或者改FreeRTOS中SVC的中断服务名称。FreeRTOS采取第二种方法，即在FreeRTOSConfig.h中添加添加宏定义的方法来修改，顺便把PendSV和SysTick的中断服务函数名也改成与向量表的一致。

```
/* Definitions that map the FreeRTOS port interrupt handlers to their CMSIS
standard names. */
#define vPortSVCHandler       SVC_Handler
#define xPortPendSVHandler    PendSV_Handler
#define xPortSysTickHandler   SysTick_Handler
```

vPortSVCHandler()函数开始真正启动第一个任务，不再返回。

```
void vPortSVCHandler( void ){
  __asm volatile (
          " ldr r3, pxCurrentTCBConst2    \n"         /* Restore the context. */
          " ldr r1, [r3]                  \n"         /* Use pxCurrentTCBConst to
get the pxCurrentTCB address. */
```

```
          " ldr r0, [r1]                  \n"         /* The first item in
pxCurrentTCB is the task top of stack. */
          " ldmia r0!, {r4-r11, r14}      \n"         /* Pop the registers and the
critical nesting count. */
          " msr psp, r0                   \n"         /* Restore the task stack
pointer. */
          " isb                           \n"
          " mov r0, #0                    \n"
          " msr basepri, r0               \n"
          " bx r14                        \n"
          "                               \n"
          " .align 4                      \n"
          "pxCurrentTCBConst2: .word pxCurrentTCB   \n"
        );
}
```

B. 任务切换

任务切换就是在就绪列表中寻找优先级最高的就绪任务，然后去执行该任务。

1. taskYIELD( )

> portYIELD 的实现很简单，实际就是将 PendSV 的悬起位置1，当没有其他中断运行的时候响应 PendSV 中断，去执行 PendSV 中断服务函数，在里面实现任务切换。

```
#define taskYIELD()         portYIELD()
/* Scheduler utilities. */
#define portYIELD()                                             \
{                                                               \
  /* Set a PendSV to request a context switch. */               \
  portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;               \
                                                \
  /* Barriers are normally not required but do ensure the code is completely within
the specified behaviour for the architecture. */  \
  __asm volatile( "dsb" ::: "memory" );                         \
  __asm volatile( "isb" );                                      \
}
```

2. xPortPendSVHandler( )

> PendSV 中断服务函数是真正实现任务切换的地方。

```
void xPortPendSVHandler( void ) {
  __asm volatile
  (
  " mrs r0, psp                 \n"
  " isb                         \n"
  "                             \n"
  " ldr r3, pxCurrentTCBConst   \n"   /* Get the location of the current TCB. */
  " ldr r2, [r3]                \n"
  "                             \n"
```

```
  " tst r14, #0x10                \n"   /* Is the task using the FPU context?  If so,
push high vfp registers. */
  " it eq                         \n"
  " vstmdbeq r0!, {s16-s31}       \n"
  "                               \n"
  " stmdb r0!, {r4-r11, r14}      \n"   /* Save the core registers. */
  " str r0, [r2]                  \n"   /* Save the new top of stack into the first
member of the TCB. */
  "                               \n"
  " stmdb sp!, {r0, r3}           \n"
  " mov r0, %0                    \n"
  " msr basepri, r0               \n"
  " dsb                           \n"
  " isb                           \n"
  " bl vTaskSwitchContext         \n"   /* NOTE: call c function here. */
  " mov r0, #0                    \n"
  " msr basepri, r0               \n"
  " ldmia sp!, {r0, r3}           \n"
  "                               \n"
  " ldr r1, [r3]                  \n"   /* The first item in pxCurrentTCB is the task
top of stack. */
  " ldr r0, [r1]                  \n"
  "                               \n"
  " ldmia r0!, {r4-r11, r14}      \n"   /* Pop the core registers. */
  "                               \n"
  " tst r14, #0x10                \n"   /* Is the task using the FPU context?  If so,
pop the high vfp registers too. */
  " it eq                         \n"
  " vldmiaeq r0!, {s16-s31}       \n"
  "                               \n"
  " msr psp, r0                   \n"
  " isb                           \n"
  "                               \n"
  " bx r14                        \n"
  "                               \n"
  " .align 4                      \n"
  "pxCurrentTCBConst: .word pxCurrentTCB  \n"
  ::"i"(configMAX_SYSCALL_INTERRUPT_PRIORITY)
  );
}
```

3. vTaskSwitchContext( )

> 选择优先级最高的任务，然后更新 pxCurrentTCB 。

```
void vTaskSwitchContext( void ) {
  if( uxSchedulerSuspended != ( UBaseType_t ) pdFALSE ){
    /* The scheduler is currently suspended - do not allow a context switch. */
    xYieldPending = pdTRUE;
  }else{
    xYieldPending = pdFALSE;

    /* Select a new task to run using either the generic C or port optimised asm
code. */
```

```
    taskSELECT_HIGHEST_PRIORITY_TASK();
  }
}

#define taskSELECT_HIGHEST_PRIORITY_TASK()                              \
{                                                                       \
  UBaseType_t uxTopPriority = uxTopReadyPriority;                       \
                                                                        \
  /* Find the highest priority queue that contains ready tasks. */      \
  while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopPriority ] ) ) ) \
  {                                                                     \
    --uxTopPriority;                                                    \
  }                                                                     \
                                                                        \
  /* listGET_OWNER_OF_NEXT_ENTRY indexes through the list, so the tasks of
\
  the same priority get an equal share of the processor time. */
\
  listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ uxTopPriority ] )
); \
  uxTopReadyPriority = uxTopPriority;
\
} /* taskSELECT_HIGHEST_PRIORITY_TASK */
```

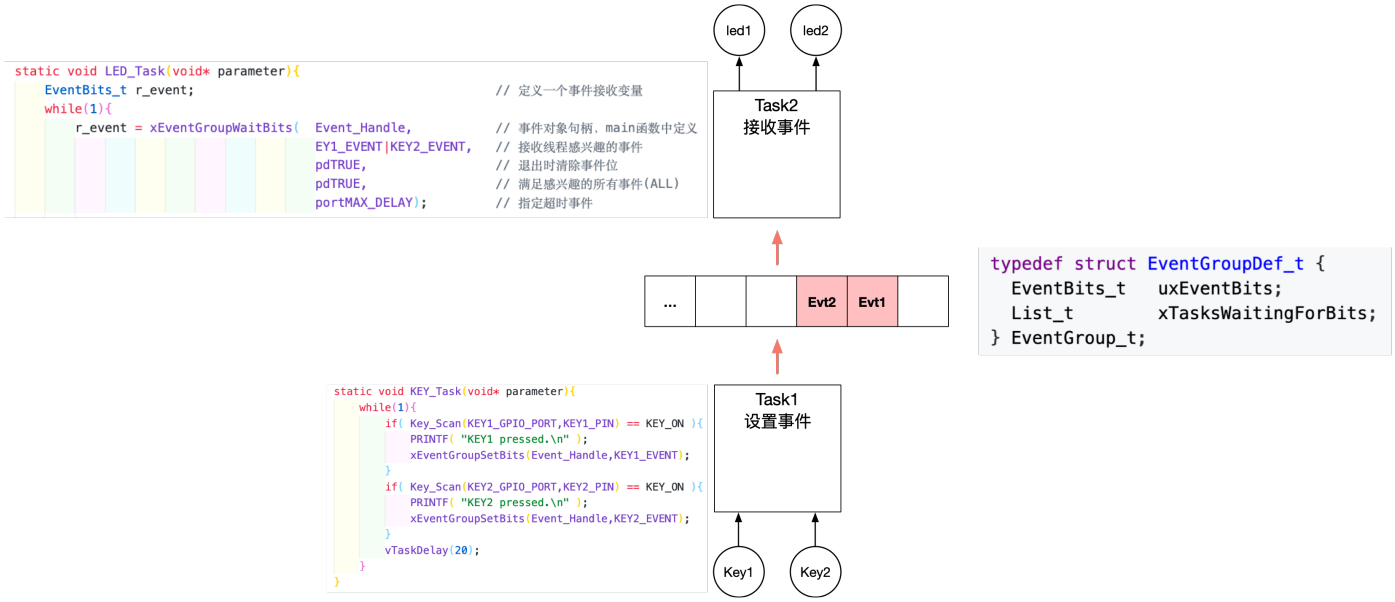
IV. 中断管理

Coming soon!

V. 事件管理

A. 概述

1. **概念：事件组（事件标志组）**
    - 事件的标志位 `uxEventBits`： 用于标识事件是否发生
    - 等待该事件组的任务列表 `xTasksWaitingForBits`： 由于等待该事件组中某个/某几个任务发生而阻塞的任务集
2. **功能：事件的等待、置位与清除**



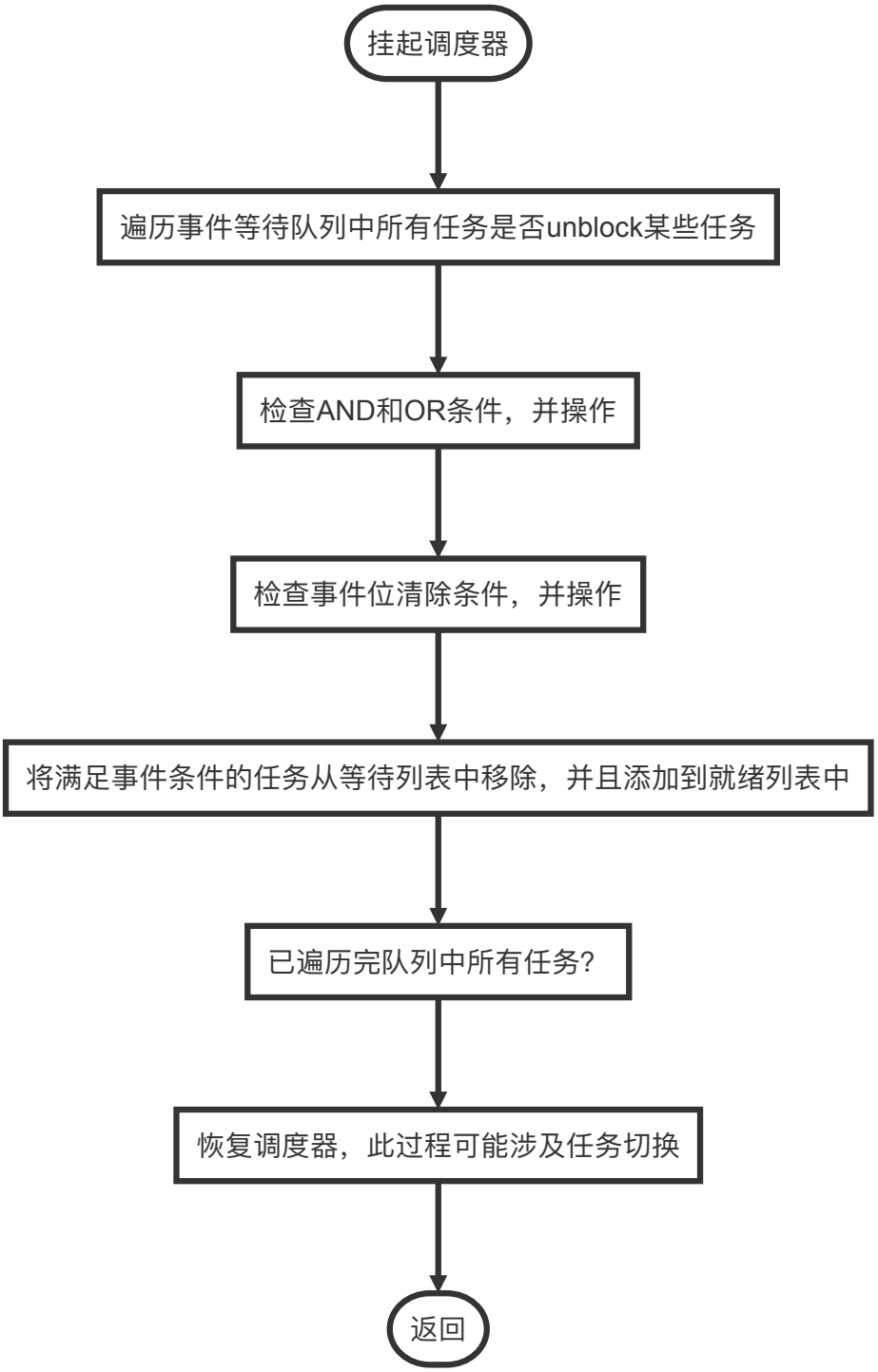

```c
static void LED_Task(void* parameter){
    EventBits_t r_event;                                    // 定义一个事件接收变量
    while(1){
        r_event = xEventGroupWaitBits(  Event_Handle,       // 事件对象句柄，main函数中定义
                                        EY1_EVENT|KEY2_EVENT,   // 接收线程感兴趣的事件
                                        pdTRUE,             // 退出时清除事件位
                                        pdTRUE,             // 满足感兴趣的所有事件(ALL)
                                        portMAX_DELAY);     // 指定超时事件
```

```c
typedef struct EventGroupDef_t {
  EventBits_t   uxEventBits;
  List_t        xTasksWaitingForBits;
} EventGroup_t;
```

```c
static void KEY_Task(void* parameter){
    while(1){
        if( Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON ){
            PRINTF( "KEY1 pressed.\n" );
            xEventGroupSetBits(Event_Handle,KEY1_EVENT);
        }
        if( Key_Scan(KEY2_GPIO_PORT,KEY2_PIN) == KEY_ON ){
            PRINTF( "KEY2 pressed.\n" );
            xEventGroupSetBits(Event_Handle,KEY2_EVENT);
        }
        vTaskDelay(20);
    }
}
```

3. 函数功能：事件的置位 `xEventGroupSetBits( ... )`

`xEventGroupSetBits()` 用于**置位事件组中指定的位**，当位被置位之后，**阻塞在该位上的任务将会被解锁**。使用该函数接口时，通过参数指定的事件标志来设定事件的标志位，然后遍历等待在事件对象上的事件等待列表，判断是否有任务的事件激活要求与当前事件对象标志值匹配，如果有，则唤醒该任务。简单来说，就是设置自己定义的事件标志位为1，并且看看有没有任务在等待这个事件，有的话就唤醒它。注意的是**该函数不允许在中断中使用**。



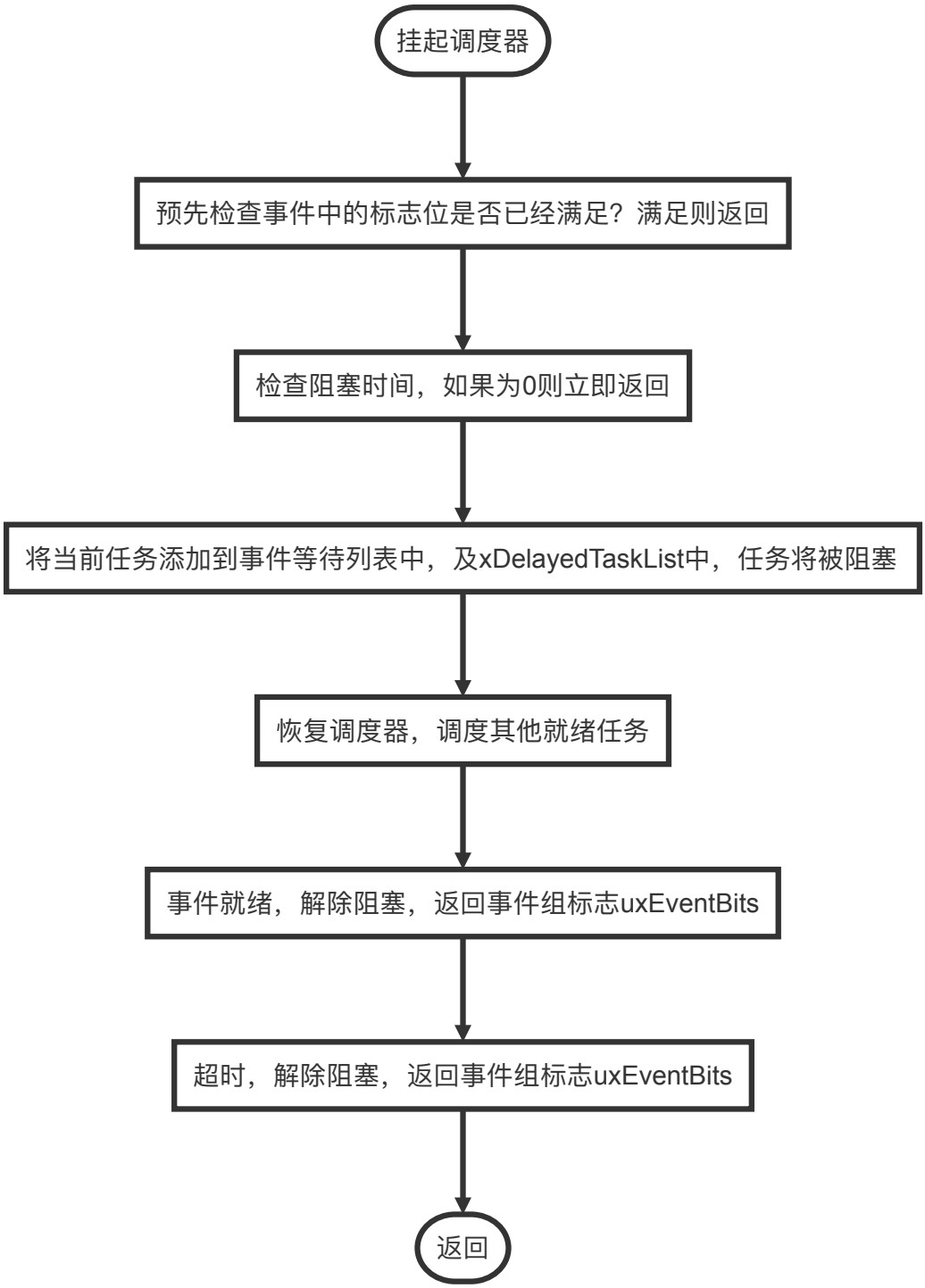
> ISSUES:
>
> 1. 每次遍历所有与该事件组相关的所有阻塞任务，时间不确定性高
> 2. 与事件相关任务的阻塞解除，主要依赖于函数 `xEventGroupSetBits( ... )` 的调用

4. **函数功能：事件的等待 `xEventGroupWaitBits( ... )`**

   `EventGroupWaitBits()` 用于获取事件组中的一个或多个事件发生标志，当要读取的事件标志位没有被置位时任务将进入阻塞等待状态。

   FreeRTOS提供了一个等待指定事件的函数 `xEventGroupWaitBits()`，通过这个函数，任务可以知道事件标志组中的哪些位，有什么事件发生了，然后通过“逻辑与”、“逻辑或”等操作对感兴趣的事件进行获取，并且这个函数实现了等待超时机制，当且仅当任务等待的事件发生时，任务才能获取到事件信息。在这段时间中，如果事件一直没发生，该任务将保持阻塞状态以等待事件发生。当其它任务或中断服务程序往其等待的事件设置对应的标志位，该任务将自

动由阻塞态转为就绪态。当任务等待的时间超过了指定的阻塞时间，即使事件还未发生，任务也会自动从阻塞态转移为就绪态。



> ISSUES:
>
> 1. 返回的是所有事件组标志uxEventBits，并不保障一定有事件发生时才有返回，用户需要自己根据语义检查
> 2. 将当前任务添加到事件等待列表时，列表是无序列表 `vTaskPlaceOnUnorderedEventList( ... )`。与 `xEventGroupSetBits( ... )` 中的查询对应。

5. **其他**

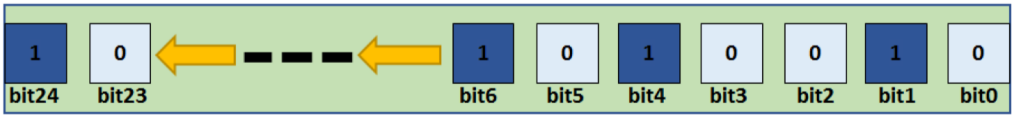
事件是一种实现任务间通信的机制，主要用于实现多任务间的同步，但事件通信只能是事件类型的通信，无数据传输。与信号量不同的是，它可以实现一对多，多对多的同步。即一个任务可以等待多个事件的发生：可以是任意一个事件发生时唤醒任务进行事件处理；也可以是几个事件都发生后才唤醒任务进行事件处理。同样，也可以是多个任务同步多个事件。

- **事件的运行机制**

  接收事件时，用户可以根据感兴趣的事件接收单个或者多个事件类型。事件接收成功后，必须使用 `xClearOnExit` 选项来清除已接收到的事件类型，否则不会清除已接收到的事件，这样就需要用户显式清除事件位。用户可以自定义通过传入参数 `xWaitForAllBits` 选择读取模式，是等待所有感兴趣的事件还是等待感兴趣的任意一个事件。

  设置事件时，对指定事件写入指定的事件类型，设置事件集合的对应事件位为1，可以一次同时写多个事件类型，设置事件成功可能会触发任务调度。

  清除事件时，根据入参数事件句柄和待清除的事件类型，对事件对应位进行 `清0` 操作。事件不与任务相关联且相互独立，一个32位的变量（事件集合，实际用于表示事件的只有24位），用于标识该任务发生的事件类型，其中每一位表示一种事件类型（0表示该事件类型未发生、1表示该事件类型已经发生），一共24种事件类型。

  事件唤醒机制，当任务因为等待某个或者多个事件发生而进入阻塞态，当事件发生的时候会被唤醒。

- **事件控制块**

  事件标志组存储在一个 `EventBits_t` 类型的变量中，该变量在事件组结构体 `EventGroup_t` 中定义。

  如果 `configUSE_16_BIT_TICKS` 定义为1，那么变量 `uxEventBits` 就是16位的，其中有8个位用来存储事件组，如果宏 `configUSE_16_BIT_TICKS` 定义为0，那么变量 `uxEventBits` 就是32位的，其中有 24 个位用来存储事件组，每一位代表一个事件的发生与否，利用逻辑或、逻辑与等实现不同事件的不同唤醒处理。除了事件标志组变量之外，FreeRTOS还使用了一个链表来记录等待事件的任务， 所有在等待此事件的任务均会被挂载在等待事件列表 `xTasksWaitingForBits` 。

```
typedef struct EventGroupDef_t {
  EventBits_t   uxEventBits;
  List_t        xTasksWaitingForBits;     /*< List of tasks waiting for a bit to be set. */
} EventGroup_t;
```

B. 事件创建函数 xEventGroupCreate()

`xEventGroupCreate()` 用于创建一个事件组，并返回对应的句柄。要想使用该函数必须在头文件 `FreeRTOSConfig.h` 定义宏 `configSUPPORT_DYNAMIC_ALLOCATION` 为1（在FreeRTOS.h中默认定义为 1）。如果使用函数 `xEventGroupCreate()` 来创建一个事件，那么需要的RAM是动态分配的。如果使用函数 `xEventGroupCreateStatic()` 来创建一个事件，那么需要的RAM是静态分配的。

- 因为**事件标志组**是FreeRTOS的内部资源，也是需要RAM的，所以在创建的时候，会向系统申请一块内存
- 如果分配内存成功，那么就对事件控制块的成员变量进行初始化，**事件标志组**变量清零，因为现在是创建事件，还没有事件发生，所以事件集合中所有位都为0，然后调用 `vListInitialise()` 函数将事件控制块中的等待事件列表进行初始化，该列表用于记录等待该事件组上的任务

在使用前需要定义一个指向该事件控制块的指针，也就是常说的事件句柄，当事件创建成功后，FreeRTOS的其他事件函数就可以根据定义的事件句柄来进行操作。如 `Event_Handle = xEventGroupCreate();`

```
EventGroupHandle_t xEventGroupCreate( void ) {
  EventGroup_t *pxEventBits;

  pxEventBits = ( EventGroup_t * ) pvPortMalloc( sizeof( EventGroup_t ) ); /*分配事件控制块的内存*/

  if( pxEventBits != NULL ) {
    pxEventBits->uxEventBits = 0;
    vListInitialise( &( pxEventBits->xTasksWaitingForBits ) );
  }
  return pxEventBits;
}
```

C. 事件删除函数 vEventGroupDelete()

D. 事件组置位函数 xEventGroupSetBits()（任务）

xEventGroupSetBits()用于**置位事件组中指定的位**，当位被置位之后，**阻塞在该位上的任务将会被解锁**。使用该函数接口时，通过参数指定的事件标志来设定事件的标志位，然后遍历等待在事件对象上的事件等待列表，判断是否有任务的事件激活要求与当前事件对象标志值匹配，如果有，则唤醒该任务。简单来说，就是设置自己定义的事件标志位为1，并且看看有没有任务在等待这个事件，有的话就唤醒它。注意的是**该函数不允许在中断中使用**。

1. **挂起调度器**，接下来的操作不知道需要多长的时间，因为需要遍历等待事件列表，并且有可能不止一个任务在等待事件，所以在满足任务等待的事件时候，任务允许被恢复，但是不允许运行，只有遍历完成的时候，任务才能被系统调度，在遍历期间，系统也不希望其他任务来操作这个事件标志组，所以暂时把调度器挂起，让当前任务占有CPU。
2. 根据用户指定的uxBitsToSet设置事件标志位
3. 设置这个事件标志位可能是某个任务在等待的事件，因此需要遍历等待事件列表中的任务
4. 获取要等待事件的标记信息，是逻辑与还是逻辑或
5. 获取任务的等待事件
6. 如果只需要有任意一个事件标志位满足唤醒任务（“逻辑或”），那么还需要看看是否有这个事件发生了？
7. 判断要等待的事件是否发生了，发生了就需要把任务恢复，在这里记录一下要恢复的任务
8. 如果任务等待的事件都要发生的时候（“逻辑与”），就需要就要所有判断事件标志组中的事件是否都发生，如果是的话任务才能从阻塞中恢复，同样也需要标记一下要恢复的任务
9. 找到能恢复的任务，然后看下是否需要清除标志位，如果需要就记录下需要清除的标志位，等遍历完队列之后统一处理。注意在一找到的时候不能清除，因为后面有可能一样有任务等着这个事件，只能在遍历任务完成之后才能清除事件标志位
10. 将满足事件条件的任务从等待列表中移除，并且添加到就绪列表中
11. 循环遍历事件等待列表，可能不止一个任务在等待这个事件
12. 恢复调度器，之前的操作是恢复了任务，现在恢复调度器，那么处于就绪态的最高优先级任务将被运行

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet ) {
ListItem_t *pxListItem, *pxNext;
ListItem_t const *pxListEnd;
List_t const * pxList;
EventBits_t uxBitsToClear = 0, uxBitsWaitedFor, uxControlBits;
EventGroup_t *pxEventBits = xEventGroup;
BaseType_t xMatchFound = pdFALSE;
```

```
  pxList = &( pxEventBits->xTasksWaitingForBits );
  pxListEnd = listGET_END_MARKER( pxList );
  vTaskSuspendAll();                                /* 1. 挂起调度器*/
  {
    pxListItem = listGET_HEAD_ENTRY( pxList );
    pxEventBits->uxEventBits |= uxBitsToSet;      /* 2. Set the bits. */

    /* 3. See if the new bit value should unblock any tasks. */
    while( pxListItem != pxListEnd ) {
      pxNext = listGET_NEXT( pxListItem );
      uxBitsWaitedFor = listGET_LIST_ITEM_VALUE( pxListItem );
      xMatchFound = pdFALSE;
      uxControlBits = uxBitsWaitedFor & eventEVENT_BITS_CONTROL_BYTES;  /* 4. 获取要等待事
件的标记信息 */
      uxBitsWaitedFor &= ~eventEVENT_BITS_CONTROL_BYTES;                /* 5. 获取任务的等
待事件 */
      /* 6. 只需要有一个事件标志位满足? */
      if( ( uxControlBits & eventWAIT_FOR_ALL_BITS ) == ( EventBits_t ) 0 ) {
        if( ( uxBitsWaitedFor & pxEventBits->uxEventBits ) != ( EventBits_t ) 0 ){  /*
7. 等待的事件是否发生? */
          xMatchFound = pdTRUE;
        }
      }
      /* 8. 否则就要所有事件都发生的时候才能解除阻塞 */
      else if( ( uxBitsWaitedFor & pxEventBits->uxEventBits ) == uxBitsWaitedFor ){
        xMatchFound = pdTRUE;
      }

      if( xMatchFound != pdFALSE ){
        /* 9. The bits match.  Should the bits be cleared on exit? */
        if( ( uxControlBits & eventCLEAR_EVENTS_ON_EXIT_BIT ) != ( EventBits_t ) 0 ){
          uxBitsToClear |= uxBitsWaitedFor;
        }
        /* 10. 将满足事件条件的任务从等待列表中移除，并且添加到就绪列表中 */
        vTaskRemoveFromUnorderedEventList( pxListItem, pxEventBits->uxEventBits |
eventUNBLOCKED_DUE_TO_BIT_SET );
      }

      /* 11. Move onto the next list item. */
      pxListItem = pxNext;
    }
    pxEventBits->uxEventBits &= ~uxBitsToClear;
  }
  /* 12. 恢复调度器 */
  ( void ) xTaskResumeAll();
  return pxEventBits->uxEventBits;
}
```

E. 事件组置位函数 xEventGroupSetBitsFromISR()（中断）

xEventGroupSetBitsFromISR()是xEventGroupSetBits()的中断版本，用于置位事件组中指定的位。 置位事件组中的标志位是一个不确定的操作，因为阻塞在事件组的标志位上的任务的个数是不确定的。FreeRTOS 不允许不确定的操作在中断和临界段中发生的，所以xEventGroupSetBitsFromISR()给FreeRTOS的软件定时器服务任务送一个消息，让置位事件组的操作在软件定时器服务任务里面完成，软件定时器服务任务是基于调度锁而非临界段的机制来实现的。

FreeRTOS的软件定时器服务任务与其他任务一样，都是系统调度器根据其优先级进行任务调度的，但软件定时器服务任务的优先级必须比任何任务的优先级都要高，保证在需要的时候能立即切换任务从而达到快速处理的目的，因为这是在中断中让事件标志位置位，其优先级由FreeRTOSConfig.h中的宏configTIMER_TASK_PRIORITY来定义。

其实xEventGroupSetBitsFromISR()函数真正调用的也是xEventGroupSetBits()函数，只不过是在软件定时器服务任务中进行调用的，所以它实际上执行的上下文环境依旧是在任务中。

要想使用该函数，必须把configUSE_TIMERS和INCLUDE_xTimerPendFunctionCall这些宏在FreeRTOSConfig.h中都定义为1。

调用过程为：anInterruptHandler()➡️xEventGroupSetBitsFromISR(...) ➡️xTimerPendFunctionCallFromISR(...)➡️xQueueSendFromISR(...)➡️xEventGroupSetBits()

F. 等待事件函数 xEventGroupWaitBits()

EventGroupWaitBits()用于获取事件组中的一个或多个事件发生标志，当要读取的事件标志位没有被置位时任务将进入阻塞等待状态。

FreeRTOS提供了一个等待指定事件的函数xEventGroupWaitBits()，通过这个函数，任务可以知道事件标志组中的哪些位，有什么事件发生了，然后通过“逻辑与”、“逻辑或”等操作对感兴趣的事件进行获取，并且这个函数实现了等待超时机制，当且仅当任务等待的事件发生时，任务才能获取到事件信息。在这段时间中，如果事件一直没发生，该任务将保持阻塞状态以等待事件发生。当其它任务或中断服务程序往其等待的事件设置对应的标志位，该任务将自动由阻塞态转为就绪态。当任务等待的时间超过了指定的阻塞时间，即使事件还未发生，任务也会自动从阻塞态转移为就绪态。

函数API使用的参数：

- 入口条件 | xEventGroup：事件句柄
- 入口条件 | uxBitsToWaitFor：指定需要等待事件组中的哪些位置1。如果需要等待bit 0 and/or bit 2那么uxBitsToWaitFor配置为0x05(0101b)
- 入口条件 | xClearOnExit：配置项。pdTRUE：当xEventGroupWaitBits()等待到满足任务唤醒的事件时，系统将清除由形参uxBitsToWaitFor指定的事件标志位；pdFALSE：不会清除由形参uxBitsToWaitFor指定的事件标志位
- 入口条件 | xWaitForAllBits：pdTRUE：“逻辑与”；pdFALSE：“逻辑或”
- 入口条件 | xTicksToWait：最大超时时间，单位为系统节拍周期
- 出口条件 | EventBits_t类型值：返回事件中的哪些事件标志位被置位，返回值很可能并不是用户指定的事件位，需要对返值进行判断再处理

函数实现逻辑：

1. 先查看当前事件中的标志位是否已经满足条件了任务等待的事件，prvTestWaitCondition()函数其实就是判断一下用户等待的事件是否与当前事件标志位一致
2. 满足条件了，就可以直接返回了，注意这里返回的是的当前事件的所有标志位，所以这是一个不确定的值，需要用户自己判断一下是否满足要求。然后把用户指定的等待超时时间xTicksToWait也重置为0，这样等下就能直接退出函数返回
3. 看看在退出的时候是否需要清除对应的事件标志位，如果xClearOnExit为pdTRUE则需要清除事件标志位，如

果为pdFALSE就不需要清除
4. 当前事件中不满足任务等待的事件，并且用户指定不进行等待，那么可以直接退出，同样也会返回当前事件的所有标志位，所以在使用`xEventGroupWaitBits()`函数的时候需要对返回值做判断，保证等待到的事件是任务需要的事件
5. 而如果用户指定超时时间了，并且当前事件不满足任务的需求，那任务就进入等待状态以等待事件的发生
6. 保存一下当前任务的信息标记，以便在恢复任务的时候对事件进行相应的操作
7. 将当前任务添加到事件等待列表中，任务将被阻塞指定时间`xTicksToWait`， 并且这个列表项的值是用于保存任务等待事件需求的信息标记，以便在事件标志位置位的时候对等待事件的任务进行相应的操作
8. 恢复调度器
9. 在恢复调度器的时候，如果有更高优先级的任务恢复了，那么就进行一次任务的切换
10. 程序能进入到这里说明当前的任务已经被重新调度了，调用`uxTaskResetEventItemValue()`返回并重置`xEventListItem`的值，因为之前事件列表项的值被保存起来了，现在取出来看看是不是有事件发生
11. 如果仅仅是超时返回，那系统就会直接返回当前事件的所有标志位
12. 再判断一次是否发生了事件
13. 如果发生了，那就清除事件标志位并且返回
14. 返回事件所有标志位

```
EventBits_t xEventGroupWaitBits( EventGroupHandle_t xEventGroup, const EventBits_t
uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits,
TickType_t xTicksToWait ) {
EventGroup_t *pxEventBits = xEventGroup;
EventBits_t uxReturn, uxControlBits = 0;
BaseType_t xWaitConditionMet, xAlreadyYielded;
BaseType_t xTimeoutOccurred = pdFALSE;

  vTaskSuspendAll();
  {
    const EventBits_t uxCurrentEventBits = pxEventBits->uxEventBits;

    /* 1. Check to see if the wait condition is already met or not. */
    xWaitConditionMet = prvTestWaitCondition( uxCurrentEventBits, uxBitsToWaitFor,
xWaitForAllBits );

    if( xWaitConditionMet != pdFALSE ){
      /* 2. The wait condition has already been met so there is no need to block. */
      uxReturn = uxCurrentEventBits;
      xTicksToWait = ( TickType_t ) 0;

      /* 3. Clear the wait bits if requested to do so. */
      if( xClearOnExit != pdFALSE ){
        pxEventBits->uxEventBits &= ~uxBitsToWaitFor;
      }
    }else if( xTicksToWait == ( TickType_t ) 0 ){
      /* 4. The wait condition has not been met, but no block time was specified, so
just return the current value. */
      uxReturn = uxCurrentEventBits;
      xTimeoutOccurred = pdTRUE;
    }else{
      /* 5. The task is going to block to wait for its required bits to be set.
uxControlBits are used to remember the specified behaviour of
      this call to xEventGroupWaitBits() - for use when the event bits unblock the task.
*/
      if( xClearOnExit != pdFALSE ){    /* 6. 保存一下当前任务的信息标记 */
        uxControlBits |= eventCLEAR_EVENTS_ON_EXIT_BIT;
      }
```

```
      if( xWaitForAllBits != pdFALSE ){
        uxControlBits |= eventWAIT_FOR_ALL_BITS;
      }

      /* 7. Store the bits that the calling task is waiting for in the task's event list
item so the kernel knows when a match is
      found.  Then enter the blocked state. */
      vTaskPlaceOnUnorderedEventList( &( pxEventBits->xTasksWaitingForBits ), (
uxBitsToWaitFor | uxControlBits ), xTicksToWait );

      uxReturn = 0;
    }
  }
  xAlreadyYielded = xTaskResumeAll();   /* 8. 恢复调度器 */

  if( xTicksToWait != ( TickType_t ) 0 ){
    if( xAlreadyYielded == pdFALSE ){
      portYIELD_WITHIN_API();           /* 9. 在恢复调度器的时候，如果有更高优先级的任务恢复
了，那么就进行一次任务的切换 */
    }
    /* 10. The task blocked to wait for its required bits to be set - at this point
either the required bits were set or the block time
    expired.  If the required bits were set they will have been stored in the task's
event list item, and they should now be
    retrieved then cleared. */
    uxReturn = uxTaskResetEventItemValue();

    /* 11. 如果仅仅是超时返回，那系统就会直接返回当前事件的所有标志位 */
    if( ( uxReturn & eventUNBLOCKED_DUE_TO_BIT_SET ) == ( EventBits_t ) 0 ){
      taskENTER_CRITICAL();
      {
        /* The task timed out, just return the current event bit value. */
        uxReturn = pxEventBits->uxEventBits;

        /* 12. It is possible that the event bits were updated between this task leaving
the Blocked state and running again. */
        if( prvTestWaitCondition( uxReturn, uxBitsToWaitFor, xWaitForAllBits ) !=
pdFALSE ){
          if( xClearOnExit != pdFALSE ){
            /* 13. 如果发生了，那就清除事件标志位并且返回 */
            pxEventBits->uxEventBits &= ~uxBitsToWaitFor;
          }
        }
        xTimeoutOccurred = pdTRUE;
      }
      taskEXIT_CRITICAL();
    }

    /* 14. The task blocked so control bits may have been set. */
    uxReturn &= ~eventEVENT_BITS_CONTROL_BYTES;
  }
  /* Prevent compiler warnings when trace macros are not used. */
  ( void ) xTimeoutOccurred;
  return uxReturn;
}
```

G. 事件组清除置位函数 xEventGroupClearBits() 与 xEventGroupClearBitsFromISR()

> `xEventGroupClearBits()`与`xEventGroupClearBitsFromISR()`都是用于清除事件组指定的位，如果在获取事件的时候没有将对应的标志位清除，那么就需要用这个函数来进行**显式清除**，`xEventGroupClearBits()`函数不能在中断中使用，而是由具有中断保护功能的`xEventGroupClearBitsFromISR()`来代替，中断清除事件标志位的操作在定时器服务任务里面完成。